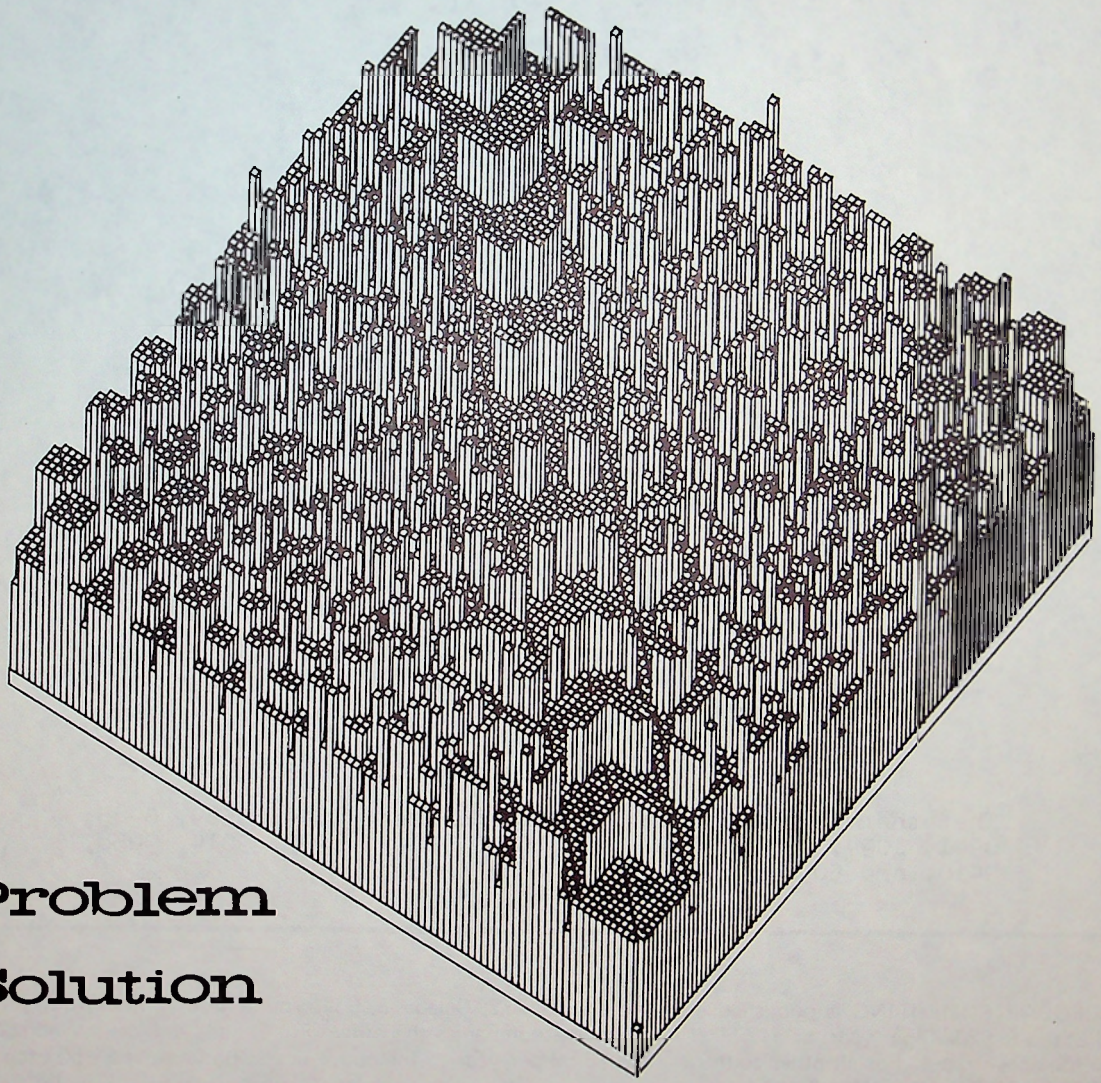
Popular Computing

53

August 1977 Volume 5 Number 8



Problem Solution

Mr. Anderson's Problem

Mr. Anderson's Problem (issue 52) was the following:

Take a 4-digit number. Repeat its high order digit at the low order end. Take the absolute differences of the adjacent digits, forming a new 4-digit number. Repeat the process until 0000 is reached, and count the steps. For the 10,000 4-digit numbers, tabulate the results. The process is shown here for the number 9850:

```
9  8  5  0  9
 1  3  5  9  1
  2  2  4  8  2
   0  2  4  6  0
    2  2  2  6  2
     0  0  4  4  0
      0  4  0  4  0
       4  4  4  4  4
        0  0  0  0
```

The number 0000 goes out in one step. With that notation, the number 9850 takes 9 steps. Of the 10,000 4-digit numbers, the tabulation is:

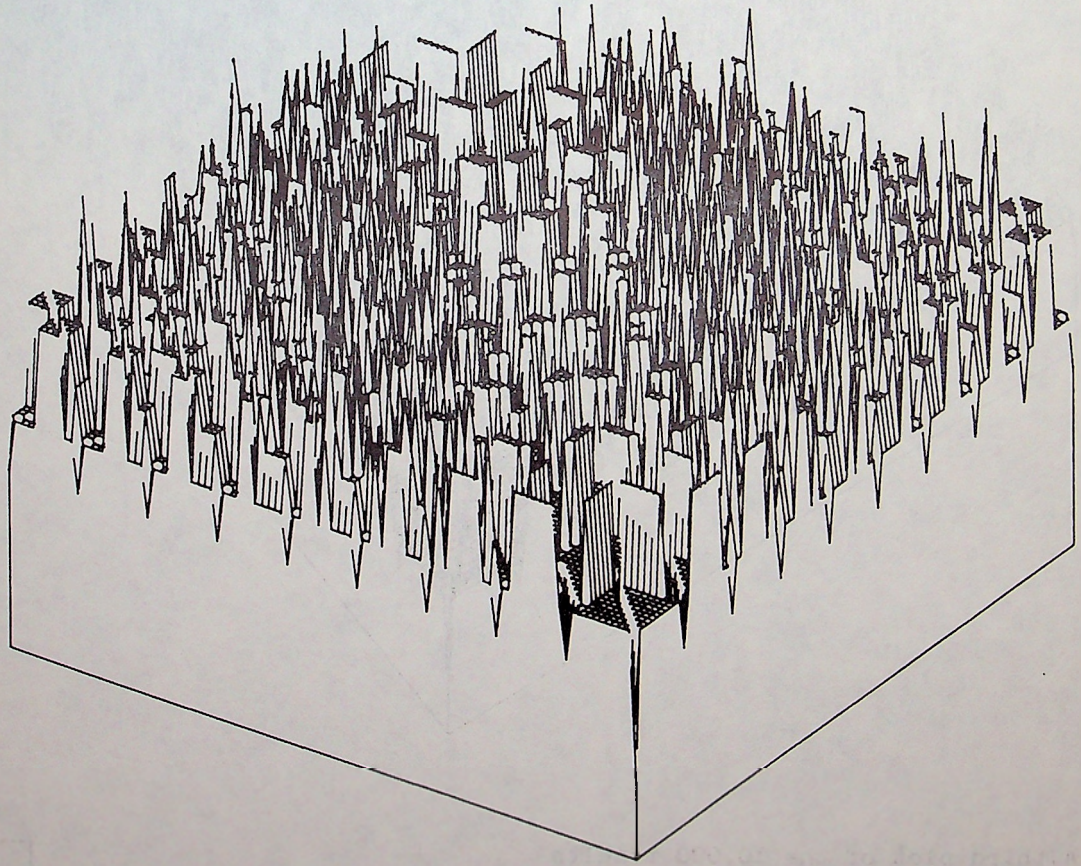
1	1	6	1152
2	9	7	1440
3	170	8	352
4	1140	9	16
5	5720		

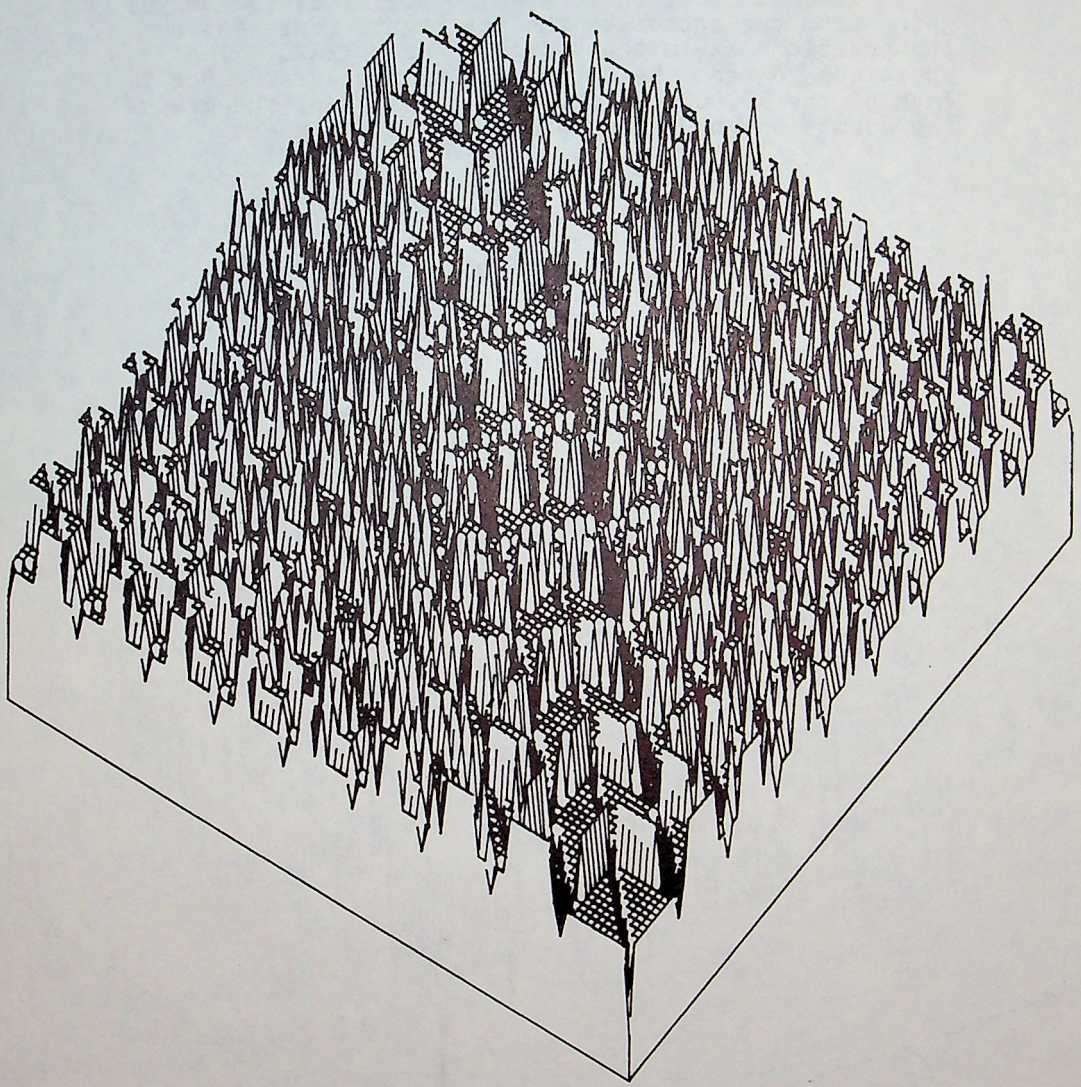
The other 15 numbers taking 9 steps are: 0149, 0589, 0985, 0941, 1094, 1490, 4109, 4901, 5890, 8509, 8905, 9014, 9058, 9410, and 5098.

POPULAR COMPUTING is published monthly at Box 272, Calabasas, California 91302. Subscription rate in the United States is $20.50 per year, or $17.50 if remittance accompanies the order. For Canada and Mexico, add $1.50 per year to the above rates. For all other countries, add $3.50 per year to the above rates. Back issues $2.50 each.

The calculations were made by Associate Editor David Babcock, who also plotted the results. Three such plots are shown, to demonstrate the versatility of good plotting routines.

In each plot, there are 100 positions (00 to 99) going from the bottom to the far right, representing the positions XX in the 4-digit number YYXX. The YY positions go from the bottom to the far left. For each of the 10,000 positions, the height of the graph represents the number of steps for that number.





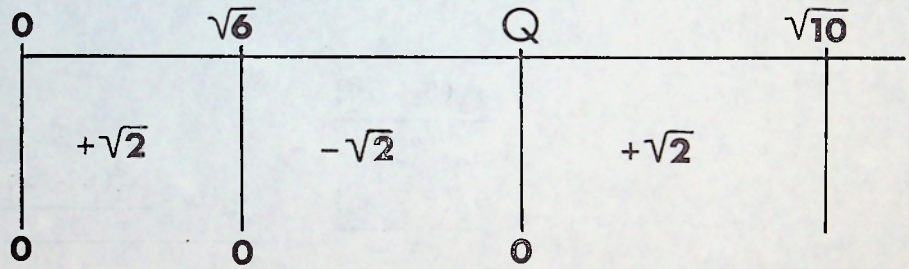
A third plot of the 10,000 results
for Mr. Anderson's problem.

That Q Value

In John Todd's article "Think Before You Cross San Pasqual," (issue 51, May), he pointed out that the recursion for square root:

$$x_{n+1} = \frac{x_n(3N - x_n^2)}{2N}$$

exhibits peculiar behavior that bears looking into. We showed it this way (for N = 2):



where the numbers at the top are values of x_0, the starting value for the recursion, and the numbers below are the values to which it converges. The problem then given was: What is the value of Q, which appears to be a sharp dividing line between starting values that lead to plus or minus the square root of 2?

Herman P. Robinson, Lafayette, California, has analyzed the problem, as follows:

"The equation we are talking about is:

$$x_{n+1} = \frac{x_n(6 - x_n^2)}{4}$$

If you start with $x_1 = 0$, the iterations stay at 0. If x_1 is nearly zero and positive, the first iteration will give a new x about 6/4 or 1.5 times the last, and the iterations continue increasing at this rate as long as x is small. The x increases monotonically, and eventually reaches a limiting positive value, if you start out with a small positive x. If you start with a small negative x, the values proceed monotonically to a limiting negative x.

If, now, you start with a relatively large x numerically greater than $\sqrt{2}$, these remarks do not hold, and large swings in the value can occur. Let the starting value be such that $6 - x^2$ rather than x is nearly zero; then x must be $\pm\sqrt{6}$, nearly. After the first iteration, the x is now at a small value and the arguments of the previous paragraph apply, and you will end eventually at a plus or minus value, depending on whether x_1^2 is less or greater than 6. Similarly, we can ask that the second iteration produce a nearly zero x. We then have:

$$x_2 = \frac{x_1(6 - x_1^2)}{4}$$

$$x_3 = \frac{x_2(6 - x_2^2)}{4} = \frac{x_2(6 - \frac{x_1^2(6 - x_1^2)^2}{16})}{4}$$

Let $\quad 6 - \frac{x^2(6 - x^2)^2}{16} = 0$

Then $\quad x^6 - 12x^4 + 36x^2 - 96 = 0$, for which the real roots are:

$\pm$ 3.03739 62088 33330 57287 46830 70252 23598 26562 42275

$= \pm\left[4 + 2(5 + \sqrt{24})^{1/3} + 2(5 - \sqrt{24})^{1/3}\right]^{1/2}$

Theoretically this procedure can be used to get an infinite number of such anomalistic constants, all smaller than $\sqrt{10}$. However, it's difficult. The next ones are

$\pm$ 3.14129 03603 49289 43052 21234 25086 01567 13528 40434,

roots of $x^2(6 - x^2)^2\left[96 - x^2(6 - x^2)^2\right]^2 = 393216$;

and

$\pm$ 3.15877 49288 57708 24110 82504 85499 84723 40040 81493 .

The final result is that there is no single value of Q but infinitely many, and $\sqrt{6}$ is the first one, with all the areas between Q's alternating in sign in the diagram.

In the sequence of numbers obtained in the iteration, if one ended at zero, each previous number would be one of the Q's. Thus, $\sqrt{6}$ is the last one before the 0; 3.03739... is the last one before $\sqrt{6}$, and so on. Thus, we can obtain any Q, given the one before it:

$$Q_{n+1}(Q_{n+1}^2 - 6) = 4Q_n$$

and the succession of Q values is:

$Q_1 = \sqrt{6}$ = 2.44948 97427 83178 09819 72841

Q_2 = 3.03739 62088 33330 57287 46831

Q_3 = 3.14129 03603 49289 43052 21234

Q_4 = 3.15877 49288 57708 24110 82505

Q_5 = 3.16169 37368 46074 56507 65943

Q_6 = 3.16218 03358 70556 04256 00058

Q_7 = 3.16226 14393 48070 28789 21324

Q_8 = 3.16227 49566 95438 78206 40148

Q_9 = 3.16227 72095 89475 65577 92208

Q_{10} = 3.16227 75850 71893 15676 12129

$\sqrt{10}$ = 3.16227 76601 68379 33199 88935" □

To keep our numbering system for problems up to date, issue 52 should have the following added:

Problem 181 (page 7) Beiler's Coin Problem
Problem 182 (page 13) Meally's Continued Product
Problem 183 (page 19) Mr. Anderson's Problem
Problem 184 (page 20) The Binary/Decimal Game

BOOK REVIEW

DATA PROCESSING FOR BUSINESS Second Edition

by Gerald A. Silver and Joan B. Silver
Harcourt Brace Jovanovich, 1977, 596 pages.

The publication of this second edition is heralded by the publisher with a 12-page brochure mailed broadside to computing departments of colleges. It is not generally recognized that book promotion follows the same rule as the promotion of movies; namely, promotion is a function of sales. This means that whatever is initially successful is promoted, and that product which is not immediately successful is allowed to die. You can verify this Law in the movie section of your daily paper. The quality film that does not score an instant success is relegated to one-inch ads so that it has no further chance of success. It's logically backwards, but there it is. There are exceptions, but they are only that...exceptions. In the book business, the exceptions are always clearly labelled "publishing phenomena."

The Silvers' book is aimed at the community college market, for first (and last) course in data processing. The treatment of the stated subject (which carefully excludes any mathematical or engineering calculations) is comprehensive but shallow. The student who learns with this text will develop a passing familiarity with most topics of business use of computers, but will know very little about computing.

He will know little, for example, about programmed loops, except for a short paragraph on DO-loops in Fortran. He will not know how to construct his own loops (say, in assembly language) and certainly not with index registers. This would be all right (such coding tricks are readily learned) provided he knew why loops are used, and when... but he will not have been told these things.

Similarly, he will have learned that subroutines are things that are built into Fortran, or that are written by manufacturers and stored in libraries. He will have missed completely the concept of using subroutines as a tool for segmenting his work.

The whole book is thus just a little bit off; just enough to form a distorted picture of what business computing is all about. It will make it difficult later, if the student decides to concentrate on computing as his major, to learn how computers are used in the real world.

For example, "byte" is defined as one vertical column of bits on a card (12-bit bytes are something new in our field). Following that, the student learns that "the computer automatically translates the holes in the card into the proper storage code." Some of these students will show up in other computing courses, all breathless with their knowledge of all the things that computers do "automatically." It's sad, because it costs little more to tell the truth--except that the authors must first know the truth.

It may be lack of knowledge, or simply a desire to make the book have a wider appeal. The authors explicitly define "computer" to include any number-processing device, including the abacus, the slide rule, and pocket calculators. They also attempt to define mini computers in terms of their rack dimensions, price, and storage capacity. Such attempts can barely get into print in time to become thoroughly obsolete.

There are little jarring notes all through the book:

> "APL programs can be run only on larger computers";
>
> "APL programs can be entered only via special terminals";
>
> "BAUDOT CODE: A code for the transmission of data in which five equal-length bits represent one character";
>
> "What Are Data and Data Processing?"

Every page of the book is printed in two colors (which is generally good) and the book contains many cartoons from the popular press (which may or may not be good, depending on whether you are appealing to children or students who want to learn).

Students exposed to this text will receive an overview of business computing. The view will be shallow and somewhat misleading, but not too much harm will be done if the student needs only a survey of the field, and does not intend to make it a career.

fg

Problem Solution

Problem 182 (issue 52, page 13), due to Victor Meally, was the following:

Form the product:

$$\frac{2}{1} \cdot \frac{3}{2} \cdot \frac{5}{4} \cdot \frac{7}{6} \cdot \frac{11}{10} \cdot \frac{13}{12} \cdot \frac{17}{16} \cdot \frac{19}{18} \cdots$$

where each factor is of the form p/(p-1), with the p's successive primes. Mr. Meally observed that whenever the product first exceeded each successive integer, the number of terms involved formed the sequence:

1, 2, 3, 4, 6, 9, 14, 22, 35,...

which bears a striking resemblance to the sequence:

0, 1, 2, 3, 5, 8, 13, 21, 34,...

To test his conjecture, it is necessary to calculate the product terms to high precision, to find at what point the product first exceeds 10.00.

Associate Editor David Babcock performed the calculations to 100 significant digits, using the EXACMATH program available at California State University, Northridge. The product first exceeds 10.0 with the fraction 257/256, which is the 55th prime. Thus, the next term in Meally's sequence is 54, rather than the 55 that had been conjectured. The product first exceeds 11.0 with the fraction 461/460 (the 89th prime).

The 100th prime gives the fraction 541/540, and the product including that fraction is:

11.26762038958268647980911350617923956596502794708771179907281756853599000396272693771 2

Our source for the table of natural logarithms (issue 51, page 15) was seriously in error. The simplest correction to make would be to delete the last five digits on every line. However, Dr. John W. Wrench, Jr. has kindly furnished the following table of the 41st through 50th decimal places of the logarithms:

51	93570 40849	70	86218 21063
52	68213 48072	71	76526 71057
53	77752 91122	72	81518 72626
54	71807 82850	73	52844 87160
55	08208 20593	74	77721 52789
56	85132 66263	75	93266 35978
57	69819 12188	76	79530 01964
58	13048 28300	77	91583 51928
59	46789 33145	78	58636 94321
		79	41224 42261
60	92180 81178	80	01891 70954
61	81743 08126	81	62231 29100
62	39064 84085	82	10603 75956
63	65845 22736	83	13056 66324
64	00806 16153	84	75556 12512
65	69299 02872	85	04366 85426
66	97546 12042	86	36319 24181
67	32903 22190	87	03471 74549
68	03281 30626	88	07257 01818
69	70872 66319	89	60387 13296

90	82604 27427
91	52674 34207
92	80583 56095
93	29488 30334
94	50636 38048
95	80615 56765
96	91229 62403
97	14902 65878
98	69593 52398
99	87969 58292
100	02977 25755

One Man's View On What Is Worth Computing

Young children, when they first learn to count, sometimes try to count to the largest number possible; they do not comprehend the futility of trying to get to infinity using finite methods. Later, some try to name the largest number possible. A few, when they learn the generality of the process of multiplication, try to do the largest multiplication possible.

Many people value uniqueness. It is very easy to do something that is probably unique. Take a 64-bit random number and multiply it by another 64-bit random number. Then, assuming that $a \cdot b = b \cdot a$, there will be

$$2^{127} = 1.70141 \cdot 10^{38}$$

possible multiplications. On the other hand, a million computers working for a million years at a million times the speed of present day computers (which is approximately a million multiplications per second) would perform

$$3.15576 \cdot 10^{31}$$

multiplications, from which it is easy to see that there is less than one chance in a million that your random muliplication would have been done. Uniqueness of a computation, then, is no test for worthwhileness.

The 6002-digit number shown in issue No. 19 of POPULAR COMPUTING is claimed to be the largest known prime number, and I don't doubt that it is. Fred Gruenberger has said that he would gladly drop all other activities if he thought he could find a 7000-digit prime. Even Euclid knew that there were an infinite number of prime numbers, and finding one more does not strike me as particularly interesting. One knows that with enough effort a larger prime can be found, and I for one believe that no great insight will arise from finding one more larger prime.

It seems to me that what we are discussing is this: When there are known to be an infinite number of similar cases (or sufficiently many so that the difference from infinity is not important), when does one decline to compute another case or even be interested in the results? Number theory has found much of its inspiration from a few special cases. For example, it was an extensive table of primes that first suggested the analytical form for the distribution of prime numbers.

And indeed, not only for numbers, but in other matters, when does one pursue an isolated, special case? The great Karl Friedrich Gauss, when asked about Fermat's last theorem, replied that he would not work on it as an isolated case but only as it fitted into a general theory.

I was forced into a similar decision some 15 or more years ago when, after computing the answer to yet another problem (and having failed to compute a few of those proposed), I saw that my life would be an endless sequence of isolated problems unless I made a decision somewhat as Gauss did. I came to the conclusion that in the future I would work on isolated problems only when viewed as special cases of more general situations. Of course, there is also an endless sequence of general theories to be developed, so in a sense the decision only raised the level a bit but did not remove the essential infinity facing me.

I have to a great extent stuck by that difficult decision to always view the proposed problem in the light of a larger class of similar problems; it has stood me in good stead. Frequently the more general view has suggested simpler methods of approaching the particular problem--a not uncommon feature of abstract mathematics! And even if the general case is less machine efficient than some special, trick method, still it is more efficient from the human's point of view when the vastness of the unknown is considered.

I believe that with an infinite sea of numbers to be computed, and an infinite number of tricks to use, we need to be prudent and try to find some degree of generality. I am not saying that special cases should not be computed, nor that occasionally finding an isolated trick for an isolated problem should not be done; it is that they should not be the major part of a life of serious work. There is a great difference between occasionally getting one's hands dirty and wallowing in the sea of computations that can be done.

The often given justification, "It amuses me," cannot be refuted on its own level, but it should be obvious that if you want society to support you, then consideration must be given to what is worth doing for others--to what society thinks it wants.

--RICHARD W. HAMMING

PC53-14

BOOK REVIEW

A DISCIPLINE OF PROGRAMMING
by Edsger W. Dijkstra

Anyone who has ever been associated with the development of a large or complex program is familiar with the tendency for such programs eventually to become so large and so complex that no one individual really understands them or can reliably predict how they will behave in some previously unencountered situation. Such programs have many of the aspects of natural systems (such as the weather or the human body), in that they cannot be characterized by simple cause-and-effect relations but instead must be understood in empirical or statistical terms. Such programs are examples of what E. W. Dijkstra calls unmastered complexity. When such complexity is encountered in natural systems, we can accept it as a fact of life. When it occurs in human artifacts such as computer programs, it is unconscionable.

The effort to master the complexity of large programs has led to the concepts of structured programming and programming by successive refinement, in whose development Professor Dijkstra has played a key role. These concepts are important because they keep the "meaning" of a program constantly before the writer and reader. Equally important to mastering complexity, but not addressed by either of these concepts, is assuring that the program is correct; that is, that it satisfies its stated objectives. In A Discipline of Programming, Professor Dijkstra describes a programming methodology which assures the development of correct programs while retaining the benefits of structured programming and programming by successive refinement.

To properly convey the significance of Dijkstra's methodology, it is necessary to review briefly some basic computer concepts:

- A computer can be viewed as a collection of variables (registers, words, etc.), each capable of holding one of a set of values.
- The state of a computer is defined by the values held by its variables.
- A computer instruction is a rule for transforming one state into another.
- A program is a sequence of instructions which transforms an initial state into a corresponding final state.

To be generally useful, a program must accommodate a set of initial states. This set can be characterized by an initial condition, i.e., a Boolean predicate in the computer variables. The set of corresponding final states produced by the program can be similarly defined by a final condition.

The programming problem consists of constructing a sequence of instructions which will transform a given initial condition into a specified final condition. Quite often the initial condition is simply the Boolean constant true, i.e., the program is to accept all possible initial states; whereas the final condition is usually much stronger. The initial and final conditions also are usually quite different from one another, so that many instructions are required to achieve the desired transformation.

The customary approach to the programming problem is to divine the form of the program on the basis of prior experience or just plain guess work, and then to drive it with enough test cases (i.e., initial states) to assure oneself that it does indeed produce the correct final states. The number of test cases considered in this process is usually a very small fraction of the total number possible; hence the programming process is generally followed by a testing phase in which the program is mechanically driven with a somewhat larger number of test cases. Even with automated testing, however, it is generally impossible to exhaust all possibilities. At some point testing must stop and the program certified "correct," even though a large number of initial states remain untried.

It is possible to prove, by means of the predicate calculus, that a program is correct for all possible input states. This is done by showing that the initial condition and the program imply the final condition; or, alternatively, that the final condition and the program imply the initial condition. Such proofs are rarely carried out in practice because the initial and final conditions are hard to formulate, the proof is tedious to carry through by hand, and automatic aids to program proving are slow in coming into general usage. However, none of these difficulties seems insurmountable. Prof. Dijkstra's book strongly suggests that the formal proof of program correctness can be achieved in practice.

In Dijkstra's methodology, a program and its proof are developed concurrently; in Dijkstra's words, "hand in hand." Thus, when the program is complete, so is its proof; it is necessary neither to test the program in the conventional manner, nor to undertake a separate, tedious formal proof. Moreover, the act of proving the program as it evolves often suggests the form the program should take; instead of being divined, the program can often be derived in an almost mechanical manner. The practice of developing a program and its proof simultaneously is the "discipline" to which the title of Dijkstra's book refers.

The methodology assumes that the semantics of the language in which the program is to be written can be defined in terms of rules for deriving the weakest initial condition which each instruction of the language will transform into a specified final condition. For example, the weakest initial condition which the assignment instruction x:=7 will transform into a given final condition R is obtained by replacing x everywhere it appears by R by the value 7. If R=(x=7 and y=10), the weakest initial conditon is

$$(7=7 \text{ and } y=10)$$
$$= (\text{true and } y=10)$$
$$= (y=10)$$

That is, it is necessary only to assure initially that y has the value 10 in order for the statement x:=7 to produce the final condition (x=7 and y=10).

Given the semantic definition of individual instructions, the semantic definition of a sequence of instructions can be obtained by functional composition. Thus, the weakest initial condition for the sequence I1;I2 is derived by

1) finding the weakest initial condition, say Q2, which I2 will transform into the desired final condition; and

2) finding the weakest initial condition Q1 which I1 will transform into Q2.

Q1 then becomes the weakest initial condition for the sequence. For example, if the sequence is x:=7;y:=10 and the desired final condition is (x=7 and y=10), the steps in the derivation are

1) Q2= (x=7 and 10=10) = (x=7)

2) Q1= (7=7) = true

that is, any initial state will lead to the desired final condition. The convention of defining instruction semantics in terms of weakest initial conditions (instead of strongest final conditions) thus leads naturally to the construction of programs in a sequence opposite to their execution sequence. This sequence of construction focuses attention on the final condition, which is generally much stronger than the initial condition, and facilitates the discovery of a correct program.

In terms of Dijkstra's methodology, the programming problem can then be stated thus: given an initial condition Q and a final condition R, find an instruction sequence IS such that Q implies the weakest initial condition defined by IS and R. For example, if Q=(x=0 and y=0) and R=(x=7 and y=10), then the statement list IS: x:=7; y:=10 is guaranteed to transform Q into R, since the weakest initial condition defined by IS and R is true (see previous paragraph), and Q implies true (in fact, any condition implies true).

To illustrate the methodology, Dijkstra introduces a simple yet powerful language composed of five instruction types: skip (do nothing), abort (stop the computer), assignment, alternative, and repetitive. The alternative instruction is a generalization of the "if-then-else" construction, containing any number of conditions or "guards" with associated instruction lists. On activation, the guards are evaluated and one of the true guards is selected at random and its instruction list executed. If none of the guards is true, the effect is the same as the abort instruction. For example, the alternative instruction

if $x \geq y \longrightarrow m:=x$

$\#y \geq x \longrightarrow m:=y$ fi

sets m to the maximum of x and y.

The repetitive instruction is a generalization of the "do while" construct. It again consists of a number of guards and associated instruction lists, but its execution consists of

a) selecting a true guard and executing its associated instruction list

b) repeating (a) as long as at least one guard is true, and then going on to the next instruction.

If no guards are initially true, the effect is the same as a skip instruction. For example, the repetitive instruction

$$\underline{\text{do}}\ \ g1 > g2 \longrightarrow g1, g2 := g2, g1$$

$$\#g2 > g3 \longrightarrow g2, g3 := g3, g2\ \ \underline{\text{od}}$$

establishes the condition $g1 < g2 < g3$.

The semantics of each instruction type are given in an expression for the weakest initial condition required to achieve a specified final condition. For all but the repetitive instruction, these expressions are sufficiently straightforward to suggest the particular instruction need in a particular situation. The expression for the repetitive instruction, however, is (as one might expect) recursive, and does not offer obvious hints as to the guards and instruction lists needed within the instruction. For this reason, Dijkstra suggests a particular strategy for formulating this instruction: find a condition which (a) holds initially or is easily established, (b) remains invariant on each repetition of the instruction, and (c) implies the desired final condition when the instruction terminates. Then the execution of the instruction is guaranteed to produce the desired final condition, if in fact it terminates. To guarantee the latter, it is necessary only to discover a positive integer function which decreases by at least one on each repetition; since the function must remain positive, the instruction must terminate. For example, the initial condition $x=0$ can be transformed into the final condition $x=10$ by the repetitive instruction

$$\underline{\text{do}}\ x < 10 \longrightarrow x := x+1\ \underline{\text{od}}$$

The invariant condition here is $x \leq 10$ and the integer function is $10 - x$.

Following the definition of his language, the author presents a number of "small formal" examples which illustrate the manner in which programs and proofs of their correctness can be developed concurrently. The language is then extended by introducing declarative constructs and array variables, and the remainder of the book is devoted almost entirely to a detailed treatment of a number of more extensive problems, such as:

- updating a sequential file
- finding the smallest prime factor of a large number
- finding the shortest subspanning tree
- finding a convex hull in three dimensions

In these problems, the emphasis is less on the methodology than on techniques which lead to successful programs, including the technique of successive refinement. These techniques are hard to codify, yet are well worth studying for the insights they provide.

Perhaps the most rewarding chapter of the book is the final one, "In Retrospect." In contrast to earlier chapters where Dijkstra addresses us as students, here he addresses us as fellow teachers. We get a glimpse of why he made the choices he did in presenting the material. His concern over unmastered complexity leads to a strategy of "separation of concerns":

> "To my taste the main characteristic of intelligent thinking is that one is willing and able to study in depth an aspect of one's subject matter in isolation,... all the time knowing that one is occupying oneself with only one of the aspects. The other aspects have to wait their turn, because our heads are so small that we cannot deal with them simultaneously without getting confused."

The trick, of course, is knowing what concerns to keep separate, or as Dijkstra expresses it (in language that would do credit to an English essayist):

> "...how to disentangle the original, amorphous knot of obligations, constraints, and goals into a set of concerns that admit a reasonably effective separation."

Two rules are offered:

1) Don't lump concerns together that are perfectly separated to start with. For example, concerns over hardware and program error are naturally separated, and should not be commingled in program design.

2) Keep concerns over program correctness separate from concerns over program efficiency. The first obligation of the programmer is correctness. Once a correct program is obtained, it can be "massaged" in such a way that it remains correct and eventually produces the requisite degree of efficiency.

Harlan Mills calls this book a "landmark in programming methodology" (Datamation, September 1976, p. 27). I agree with this assessment. While the material may be standard fare for contemporary computer scientists, it has taken a Dijkstra to make it readily accessible and to cast it into a form which is both challenging and entertaining. The student or teacher who would improve his grasp of programming will find this a rewarding book.

--reviewed by W. C. McGee

Problem Solution

Problem 114 (PC34-13) was the following:

Given a fund of $20,000 which earns monthly interest. The first month's rate is 6%, and the interest rate increases by .0000833333 each month. What constant amount can be paid out each month, so as to deplete the fund in 139 months?

On the assumption that the interest earned each month is added to the fund before the monthly payment is made, the amount of the monthly payment is $203.67. (After 139 months, the fund would still have $1.17 left.